# 3

# VBASICA Statements, Commands, and Functions

This chapter describes the VBASICA statements, commands, and functions. It is organized as follows:

**FORMAT:**

Shows the correct format for the instruction.

**PURPOSE:**

Tells what the instruction does.

**REMARKS:**

Describes in detail how to use the instruction.

**EXAMPLE:**

Shows sample programs or program segments that demonstrate the use of the instruction.

Wherever the format for a statement or command is given, the following rules apply:

1. Items in uppercase letters must be input as shown.
2. Items in lowercase letters enclosed in angle brackets ( < > ) are to be supplied by the user.
3. Items in square brackets ([ ]) are optional.

4. All punctuation except angle brackets and square brackets (that is, commas, parentheses, semicolons, hyphens, equal signs) must be included where shown.
5. Items followed by an ellipsis (...) can be repeated any number of times (up to the length of the line).

# ABS Function



**FORMAT:**

**ABS(X)**

**PURPOSE:**

Returns the absolute value of the expression X.

**EXAMPLE:**

```
PRINT ABS (7*(-5))
 35
Ok
```

# ASC Function

**FORMAT:**

**ASC(X$)**

**PURPOSE:**

Returns a numeric value that is the ASCII code of the first character of string X$. If X$ is null, VBASICA returns an "illegal function call" error. (See Appendix C for ASCII codes.)



See the CHR$ function for ASCII-to-string conversion.

**EXAMPLE:**

```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
Ok
```

# ATN Function

**FORMAT:**

**ATN(X)**

**PURPOSE:**

Returns the arctangent of X in radians. The result is from $-pi/2$ to $pi/2$. X can be any numeric type, but ATN is always evaluated in single-precision.

**EXAMPLE:**

```
10 INPUT X
20 PRINT ATN(X)
RUN
?3
 1.249046
Ok
```

3

# AUTO Command

**FORMAT:**

**AUTO [ < line > [, < increment > ]]**

**PURPOSE:**

Generates a line number automatically after each carriage return.

**REMARKS:**

< line > is the line number of the first program line.

< increment > is the number by which AUTO increments each subsequent line number.

AUTO starts numbering at the specified line number and increments each subsequent line number by the specified increment. The default for both parameters is 10. If a comma follows < line > and < increment > is not specified, VBASICA uses the increment specified in the most recent AUTO command.

If AUTO generates a line number already in use, an asterisk appears after the number to warn you that any input replaces the existing line. Press Return immediately after the asterisk to save the line and generate the next line number.

Type a CTRL-C to terminate AUTO and return to the VBASICA command level. VBASICA does not save the line in which the CTRL-C occurs.

**EXAMPLE:**

```
AUTO 100, 50
```

generates line numbers 100, 150, 200, and so on, in increments of 50.

```
AUTO
```

generates line numbers 10, 20, 30, 40, and so on, in increments of 10, the default increment.



## BEEP Statement

**FORMAT:**

**BEEP**

**PURPOSE:**

The BEEP statement sounds the speaker at 800 Hz for ¼ second.

**REMARKS:**

Both BEEP and PRINT CHR$(7); have the same effect.

**EXAMPLE:**

```
2430 IF X < 20 THEN BEEP 'if X is out of range,
     'complain.
```

# BLOAD Command

**FORMAT:**

**BLOAD** <**filespec**> [,<**offset**>]

**PURPOSE:**

Loads a memory image into memory.

3

**REMARKS:**

<filespec> is a string expression representing the file specification. In VBASICA 2.0 and later, it can contain a path as described in Chapter 1.5, except for the extension. If you omit the device name, VBASICA assumes the current drive. The only valid extensions are the following:

| | |
|---|---|
| (none) | (no extension) |
| .B | for VBASICA programs in the internal format (created with the SAVE command) |
| .P | for protected VBASICA programs in the internal format (created with the SAVE ,P command) |
| .A | for VBASICA programs in ASCII format (created with the SAVE ,A command) |
| .M | for memory image files (created with the BSAVE command) |
| .D | for data files (created by OPEN followed by output statements) |

If no period appears in the filename and if the filename has less than nine characters, VBASICA assigns the default extension .BAS.

< offset > is a numeric expression from 0 to 65535. The expression is the address at which the loading starts, specified as an offset into the segment declared by the last DEF SEG statement. If you omit the < offset >, VBASICA assumes the < offset > specified in the last BSAVE.

**WARNING:** BLOAD does not perform address range checking. Therefore, it is possible to BLOAD anywhere in memory. Ensure that you are not overwriting the operating system, VBASICA, or your own program.

**EXAMPLE:**



```
10 'Load an assembly program into VBASICA DS
20 'assuming no program has been loaded.
30 DEF SEG   'set the data segment to VBASICA's
40 BLOAD "MOVE",0 'load the CALLable program
```

# BSAVE Command

**FORMAT:**

**BSAVE < filespec >, < offset >, < length >**

**PURPOSE:**

Saves portions of the computer's memory on the specified device.

**REMARKS:**

< filespec > is a string expression representing the file specification. In VBASICA 2.0 and later, it can contain a path. Refer to Chapter 1.5 for more information on file specifications.

< offset > is a numeric expression from 0 to 65535. This expression is the address at which the saving starts, specified as an offset into the segment the last DEF SEG declared.

< length > is a valid numeric expression returning an unsigned integer from 1 to 65535, which is the length of the memory image to be saved.

**EXAMPLE:**

```
10 'To save a hi-res VICTOR screen image to disk
20 DEF SEG = &HF000  'point to VICTOR screen ram
30 HIGH.BYTE = PEEK (1) AND 7  'get lst byte of
   pointer to dot memory and mask off attribute bits
40 LOW.BYTE = PEEK(0)  'get second byte of pointer
50 HIGH.BYTE = HIGH.BYTE * 256  'shift high byte
   over four places
60 POINTER = HIGH.BYTE + LOW.BYTE  'and add
   together to make pointer
70 POINTER = POINTER * 32  'crt controller shifts
   this 5 places to get actual address, so do we
80 DOT.SEG = INT(POINTER/16)-1  'make a segment
   value
90 DEF SEG = DOT.SEG  'and go there. This is the
   memory that holds the screen image.
100 BSAVE "MYSCREEN.M",0,40000!  'screen is 400 x
    800 = 320,000 bits or 40K bytes (320,000/8).
```

# CALL Statement

**FORMAT:**

**CALL** <**variable name**> [ (<**argument list**>) ]

**PURPOSE:**

The CALL statement is the recommended way of interfacing 8086 machine language programs with VBASICA. Do not use the outmoded user call: x = USR(n).



**REMARKS:**

<variable name> contains the address of the starting point in memory of the subroutine being CALLed.

<argument list> contains the variables or constants, separated by commas, to be passed to the routine.

When you invoke the CALL statement, the following occurs:

1. For each parameter in the argument list, the 2-byte offset into the data segment [DS] of the parameter's location is pushed onto the stack.
2. The return address code segment [CS] and offset [IP] are pushed onto the stack.
3. Control is transferred to your routine via the segment address given in the last DEF SEG statement and offset given in <variable name>.

Your routine now has control. You can reference parameters by moving the stack pointer [SP] to the base pointer [BP] and adding a positive offset to [BP].

**RULES:**

The assembly language subroutine must follow these rules to work correctly:

1. It must be declared FAR.
2. Segment registers DS and ES must be restored to their entry values before returning to VBASICA.
3. The general purpose registers (AX, BX, CX, DX, SI, DI, and BP) can have any value when returning to VBASICA.
4. The assembly language routine must not change the length of any VBASICA strings.
5. The assembly language routine must perform a RET (n), where n = 2 times the number of parameters, to restore the stack pointer to its proper value.
6. You can return values to VBASICA by passing a parameter in which the result will be returned.



### *VBASICA Data Types*

To manipulate data passed to an assembly language subroutine, you must understand how the various data types are represented in memory. When a subroutine is called, VBASICA passes the address of one of the following data representations:

1. Integer: A two-byte, two's-complement number.
2. Single precision number: A four-byte, binary, floating-point quantity. The most significant byte contains the value of the exponent minus 127. The remaining three bytes contain the mantissa. The most significant byte of the mantissa contains the sign bit, followed by the seven highest bits of the mantissa. A positive number is represented with a 0 as the sign bit, and a negative number with a 1 as the sign bit. The decimal point is left of the most significant bit of

the mantissa. A 1 is always assumed to exist immediately to the right of the mantissa, although it is not represented. Thus the number is represented as the following:

```
((sign) 1.(mantissa)*2)^(exponent-127)
```

3. Double precision number: An eight-byte, binary, floating-point quantity. It is represented exactly as a single precision number, except that the mantissa consists of 55 bits, that is, 7 bytes less the sign bit.

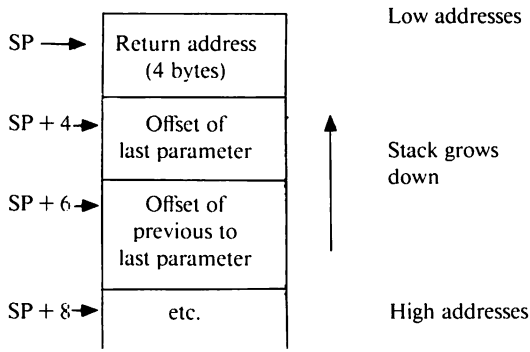
4. String: VBASICA passes the offset address of a string descriptor, which is a three-byte data structure. The first byte of the string descriptor contains the length of the string. The next two bytes contain the address of the actual ASCII string. The assembly language subroutine can modify the string, but it must not change the string descriptor or the string length.

5. Array: Arrays consist of sequential elements of the array type. For example, an integer array containing twenty elements is represented as twenty sequential integers in memory.



## *Passing Parameters*

VBASICA passes all subroutine parameters by reference—that is, the actual location of the parameter is passed, not a copy of its value. In a CALL statement, the offset of each parameter's address is pushed onto the stack in the same order that the parameters are listed in the procedure call. All parameters to the assembly language subroutine must be variables. Upon entry to the subroutine, the stack is arranged as shown in the following diagram. Reference the parameters by using the BP register to get their address off the stack.






**EXAMPLE:**

The following example shows how to load an assembly language subroutine from a VBASICA program. The assembly language routine performs modulo arithmetic on two integers, returning the remainder from dividing the first integer by the second. In this example, the assembly language module is loaded at address 1664:0 Hex, but this address is different for different applications. An explanation of the method to determine this address follows the example.

```
10 '
20 ' load the MODULO routine
30 '
40 DEF SEG = &H1664
50 BLOAD "MODULO",0
60 MODULO = 0
70 '
80 ' call the MODULO routine with some sample data
90 '
100 A% = 140
110 B% = 11
120 REMAINDER% = 0
130 CALL MODULO(A%,B%,REMAINDER%)
140 PRINT A%;"modulo";B%;"is";REMAINDER%
150 END
```

Assembly language module for use with CALL statement:

```
name        modulo

code        segment public 'code'
assume      cs:code,ds:code

modulo      proc      far

; This module is called from VBASICA with 3
; parameters, using the CALL statement.  It
; divides the first parameter by the second
; and returns the remainder in the third.

            mov     bp, sp      ;BP used to get parameters
            mov     bx, [bp+8]  ;BX=pointer to dividend
            mov     ax, [bx]    ;AX=value of dividend
            mov     bx, [bp+6]  ;BX=pointer to divisor
            mov     cx, [bx]    ;CX=value of divisor
            mov     dx, 0       ;DX:AX=dividend
            idiv    cx          ;AX=quotient,DX=remainder
            mov     bx, [bp+4]  ;BX=pointer to result
            mov     [bx], dx    ;return result to VBASICA
            ret     6           ;no. of parameters*2=6

modulo      endp
code        ends
            end
```

### *Loading the Assembly Language Module*

To call the assembly language module, you must know its location (address). With the BLOAD statement, you can load the module at any physical address. To use the BLOAD statement to load a module, you must first create the disk file containing the module with LINK, DEBUG, and the BSAVE statement, as follows:

1. After assembling your module to create the object file, use the linker to create the .EXE file. Use the /HIGH switch when linking so the module loads in high address memory.

2. Use the debugger to load the .EXE file produced in step 1.

3. Display the register values with the R command to determine where the subroutine was loaded. Write down the values contained in the CS:IP register pair and the CX register. The CS:IP register pair contains the starting address of the subroutine and the CX register contains its length.

4. Load and execute VBASICA from DEBUG with this sequence of commands:

```
NVBASICA.EXE
L
N
G
```

Your assembly language module is still loaded in high address memory.

5. Set the segment value in VBASICA with a DEF SEG statement:

**DEF SEG = <value in CS register>**

These values are hexadecimal and must be preceded with &H.

6. Save the module with a BSAVE statement:

**BSAVE <filespec>, <value in IP reg.>, <value in CX reg.>**

You can now call the assembly language subroutine from your VBASICA program. Your VBASICA program requires the following statements before you can call the subroutine:

```
DEF SEG = <value in CS register>
BLOAD <filespec>, <value in IP register>
<SUBROUTINE> = <value in IP register>
```

You can then call the subroutine with statements of the form:

**CALL <SUBROUTINE> <PARAMETER1, PARAMETER2>, ...**

NOTE: You may need to use the /M: switch to set the top of VBASICA's DS at your CS-1 to keep from loading your routine on top of VBASICA.

# CDBL Function

**FORMAT:**

**CDBL(X)**

**PURPOSE:**

Converts X to a double-precision number.

**EXAMPLE:**

```
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
 454.67    454.6699829101563
Ok
```



# CHAIN Statement

**FORMAT:**

**CHAIN [MERGE] < filespec > [,[ < line number expr > ][,ALL]**
**[,DELETE < range > ]]**

**PURPOSE:**

Calls a program and passes variables to it from the current program.

**REMARKS:**

< filespec > is a string expression representing the file specification. In VBASICA 2.0 and later, it can contain a path. Refer to Chapter 1.5 for more information on file specifications.

< line number expr > is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If you omit < line number expr >, execution starts at the first line.

< line number expr > is not affected by a RENUM command. With the ALL option, VBASICA passes every variable in the current program to the called program. If you omit the ALL option, the current program must contain a COMMON statement to list the passed variables.

If you use the ALL option, and not < line number expr >, a comma must hold the place of < line number expr >.

**EXAMPLE:**

```
CHAIN"PROG1",1000,ALL
```

If you include the MERGE option, VBASICA brings a subroutine into the program as an overlay. That is, a MERGE operation occurs with the current program and the called program. The called program must be an ASCII file if it is to be MERGEd. For example,

```
CHAIN MERGE"OVRLAY",1000
```

Delete an existing overlay each time a new overlay is brought in with the DELETE options. For example,

```
CHAIN MERGE"OVRLAY2",100, DELETE 1000-5000
```

The RENUM command affects the line numbers in < range >.

If you omit the MERGE option, CHAIN does not preserve variable types or user-defined functions for use by the chained program. You may restate any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEFFN statements containing shared variables in the chained program.

The CHAIN statement with the MERGE option leaves the files open and preserves the current OPTION BASE setting.

The VBASICA compiler does not support the ALL, MERGE, DELETE, and < line expr > options to CHAIN. Use the statement format CHAIN < filename >. If you want to maintain compatibility with the VBASICA Compiler, use COMMON to pass variables and do not use overlays. The CHAIN statement leaves the files open during CHAINing.

When you use the MERGE option, put user-defined functions ahead of any CHAIN MERGE statements in the program. Otherwise, the user-defined functions are undefined after the merge is complete.



## CHDIR Command

**FORMAT:**

**CHDIR < pathname >**

**PURPOSE:**

Changes the current directory, for VBASICA 2.0 and later releases.

**REMARKS:**

< pathname > is a string expression specifying the name of the new directory. The string expression can be a control or variable. CHDIR works exactly like the DOS command CHDIR. The < pathname > must be a string of less than 63 characters.

**EXAMPLE:** This line makes SALES the current directory:

```
CHDIR "SALES"
```

This line changes the current directory to USERS on drive B. It does NOT, however, change the default drive to B:

```
CHDIR "B:USERS"
```

The following lines change the current directory to the directory DATA one level below the root directory on the current drive:

```
PATH$="\DATA"
CHDIR PATH$
```

Also see the MKDIR and RMDIR statements. See the *DOS 2.1 Reference* for a complete discussion of directories.



## CHR$ Function

**FORMAT:**

**CHR$(I)**

**PURPOSE:**

Returns a string whose one element has ASCII code I. Refer to Appendix C for a listing of ASCII codes. CHR$ is commonly used to send a special character to the screen. For example, you can send the BEL character (CHR$(7)) as a preface to an error message, or you can send a form feed (CHR$(12)) to clear a screen and return the cursor to the home position.

See the CHR$ function for ASCII-to-numeric conversion.

**EXAMPLE:**

```
PRINT CHR$(66)
 B
Ok
```

# CINT Function

**FORMAT:**

**CINT(X)**

**PURPOSE:**

Converts X to an integer by rounding the fractional portion. An "Overflow" error occurs if X is not in the range −32768 to 32767.

Use CDBL and CSNG to convert numbers to the double- and single-precision data type. Use FIX and INT to return integers.



**EXAMPLE:**

```
PRINT CINT (45.67)
 46
Ok
```

# CIRCLE Statement

**FORMAT:**

**CIRCLE (< xcenter >, < ycenter >), < radius >**
**[, < attribute > [, < start >, < end > [, < aspect > ]]]**

**PURPOSE:**

Draws an ellipse with center (< xcenter >, < ycenter >) and radius < radius > for graphics only.

**REMARKS:**

< xcenter > is an integer expression for the x-coordinate of the center of the ellipse.

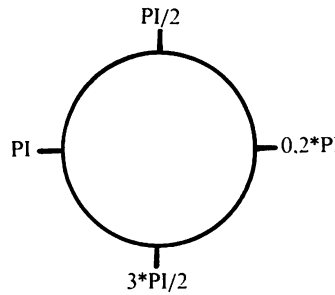
< ycenter > is an integer expression for the y-coordinate of the center of the ellipse.

< radius > is an integer expression for the radius of the ellipse.

< attribute > is an expression returning the value 0 to 3; the value determines the color of the ellipse. An attribute of 0 draws an ellipse of the background color.

< start > and < end > specify where the drawing of the ellipse begins and ends. The angles are positioned in the standard mathematical way, with 0 to the right and going counterclockwise:






If the start or end angle is negative ( − 0 is not allowed), the ellipse is connected to the center point with a line. The angles are treated as if positive (not the same as adding 2*PI). The start angle can be greater or less than the end angle.

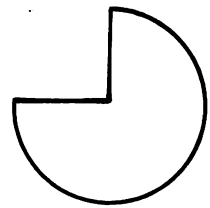
VBASICA clips points off the screen.

**EXAMPLE:**

The following example:

```
10 PI=3.14593
20 SCREEN 1
30 CIRCLE (160,100),60,,-PI,-PI/2
```

draws a part of a circle similar to the following:





<aspect> is the ratio of the x radius to the y radius. The default aspect ratio is 5.25/8.00 in high-resolution, and gives a visual circle, assuming a standard screen aspect ratio.

# CLEAR Command

**FORMAT:**

**CLEAR [,[ < expr1 > ][, < expr2 > ]]**

**PURPOSE:**

Sets all numeric variables to zero, all string variables to null, and closes all open files. CLEAR optionally sets the end of memory and the amount of stack space.



< expr1 > is a memory location that sets the highest location available for use by VBASICA.

< expr2 > sets aside stack space for VBASICA. The default is 256 bytes or one-eighth of the available memory, whichever is smaller.

**REMARKS:**

VBASICA allocates string space dynamically. An "Out of string space" error occurs if no free memory is left.

VBASICA supports the CLEAR statement with the restriction that < expr1 > and < expr2 > must be integer expressions. If you give a value of 0 for either expression, VBASICA uses the appropriate default. The default stack size is 256 bytes, and the default top of memory is the current top of memory.

With VBASICA, the CLEAR statement:

- Closes all files.
- Clears all COMMON and user variables.
- Resets the stack and string space.
- Releases all disk buffers.

**EXAMPLES:**

```
CLEAR

CLEAR ,32768

CLEAR ,,2000

CLEAR ,32768,2000
```



## CLOSE Statement

**FORMAT:**

**CLOSE[[#] < filenum > [,[#] < filenum ... > ]]**

**PURPOSE:**

Concludes I/O to a disk file.

**REMARKS:**

< filenum > is the number under which the file was OPENed. A CLOSE without arguments closes all open files.

The association between a particular file and file number ends when you execute a CLOSE. You can open the file with the same or a different file number, and you can reuse the file number to OPEN any file.

With sequential output files, a CLOSE writes the final buffer of output.

The END statement and the NEW command CLOSE all disk files automatically. STOP does not close disk files.

# CLS Statement

**FORMAT:**

**CLS**

**PURPOSE:**

Erases the current active screen and locates the cursor at the upper left corner of the screen.

3

**REMARKS:**

1. In Mode 0, the monochrome screen is cleared to white, black, or underlined, depending on the current background color. The color screen is cleared to the current background color.
2. You can also clear the screen by pressing the CTRL-L key.
3. NOTE: The SCREEN and WIDTH statements force a screen clear if the resultant screen mode created differs from the current mode.

# COLOR Statement

***Screen Mode 0***

**FORMAT:**

**COLOR [ < foreground > ] [, [ < background > ] [, < border > ]]**

**PURPOSE:**

The COLOR statement selects the foreground, background and border colors.

**REMARKS:**

< foreground > is an unsigned integer from 0 to 31 that determines the character color, with values greater than 15 blinking.

< background > is an unsigned integer from 0 to 7 that determines the color over which the character is placed. < border > is an unsigned integer from 0 to 15 that determines the color around the border of the screen, if you are using a color screen. If you are using the standard screen, VBASICA ignores this parameter.

On a color screen, COLOR selects colors according to Table 3-1.



***Table 3-1: Colors for Color Screen***

| LOW INTENSITY | COLOR | HIGH INTENSITY | COLOR |
|---|---|---|---|
| 0 | black | 8 | dark gray |
| 1 | blue | 9 | high-intensity blue |
| 2 | green | 10 | high-intensity green |
| 3 | cyan | 11 | high-intensity cyan |
| 4 | red | 12 | high-intensity red |
| 5 | magenta | 13 | high-intensity magenta |
| 6 | brown | 14 | yellow |
| 7 | white | 15 | intense white |

If < foreground > is less than 8, low-intensity colors are displayed; otherwise, high-intensity colors are displayed.

On the standard screen, the COLOR statement selects reverse video (black on white), underlined, or highlighted characters.

Table 3-2 shows the effects you can obtain with the specified combinations of foreground and background colors.

*Table 3-2: Foreground/Background Combinations*

| FOREGROUND | BACKGROUND | EFFECT |
|---|---|---|
| 7 | 0 | Normal, white on black |
| 1 | 0 | Underline, white on black |
| 0 | 7 | Reverse, black on white |
| 15 | 0 | Highlight, white on black |
| 9 | 0 | Highlight, underline, white on black |
| 8 | 7 | Reverse highlight |
| 0 | 0 | Invisible (black) |



1. Any values entered outside these ranges result in the "Overflow" or "Illegal function call" error. COLOR retains previous values.
2. You can omit any parameter. Omitted parameters assume the previously selected value.
3. The COLOR statement cannot end with a comma. If it does, a "Syntax Error" will result.
4. In Alpha mode, foreground color values 0 through 7 select character color, values 8 through 15 set the intensity bit, and values 16 through 31 set the blink bit for the character.

In Screen Mode 0, executing the COLOR statement affects only the colors of subsequently written characters.

**EXAMPLE:**

```
100 COLOR 0,7 'reverse video
110 COLOR ,0  'invisible characters
```

### *Screen Mode 1*

**FORMAT:**

**COLOR [ < background > , < palette > ]**

**PURPOSE:**

Selects the colors displayed on the screen.

**REMARKS:**

NOTE: In this mode, the color statement does not affect the standard screen.



For the color screen, the various drawing statements (PSET, LINE, and so on) allow you to specify a color attribute from 0 through 3. Color attribute 0 requests the background color, and color attributes 1–3 request foreground colors. The COLOR statement determines how these numbers are mapped to actual colors on the screen.

< background > is an unsigned integer from 0 to 15. It determines the background color and the intensity of the display according to Table 3-1.

< palette > is either 0 or 1. If < palette > is 0, or even, the foreground colors are the following:

| Color Attribute | Color |
|---|---|
| 1 | 2 (or 10) green |
| 2 | 4 (or 12) red |
| 3 | 6 (or 14) brown |

If < palette > is 1, or odd, the foreground colors are the following:

| Color Attribute | Color |
|---|---|
| 1 | 3 (or 11) cyan |
| 2 | 5 (or 13) magenta |
| 3 | 7 (or 15) white |

In Screen Mode 1, executing the COLOR statement immediately affects the colors on the entire screen.

In this mode, COLOR selects the background color and a three-color palette, any of which can be used with the graphics statements PSET, PRESET, CIRCLE, LINE, PAINT, and DRAW.

**EXAMPLE:**

```
120 COLOR 4,0      'red background,
                   green/red/yellow foreground
```

### *Screen Mode 2*

The COLOR statement is illegal in this mode.

# COM Statement

**FORMAT:**

**COM( < n > ) ON**

**COM( < n > ) OFF**

**COM( < n > ) STOP**

**PURPOSE:**

Enables or disables trapping of communications activity to the indicated serial port.



**REMARKS:**

You must execute a COM( < n > ) ON statement to allow trapping by the ON COM( < n > ) statement. After COM( < n > ) ON, if you specify a nonzero line number in the ON COM( < n > ) statement, every time VBASICA starts a new statement it checks if any characters have been input from the serial port.

If COM( < n > ) is OFF, no trapping takes place and the event is not remembered even if it does take place.

If a COM( < n > ) STOP statement is executed, no trapping can take place. If any data comes in through the serial port, it is remembered and an immediate trap occurs when COM( < n > ) ON is executed.

**EXAMPLE:**

```
10 PORT.A = 1
20 COM(PORT.A) ON  'enable com trapping on port a
    .
    .
    .
100 COM(PORT.A) STOP 'temporarily disable trapping
    .
    .
    .
300 COM(PORT.A) ON 'enable trapping immediately
    .
    .
    .
500 COM(PORT.A) OFF 'disable trapping & forget events
```



## COMMON Statement

**FORMAT:**

**COMMON** <**varlist**>

**PURPOSE:**

Passes variables to a CHAINed program.

**REMARKS:**

Use the COMMON statement with the CHAIN statement. Although COMMON statements can appear anywhere in a program, put them at the beginning.

The same variable cannot appear in more than one COMMON statement. Specify array variables with a pair of parentheses at the end of the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

**EXAMPLE:**

```
100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
      .
      .
      .
```

## CONT Command



**FORMAT:**

**CONT**

**PURPOSE:**

Continues program execution after you type a CTRL-C or after VBASICA executes a STOP or END statement.

**REMARKS:**

Execution resumes at the point where the break occurs. If the break occurs after a prompt for an INPUT statement, the program resumes execution by displaying the prompt or prompt string.

CONT is usually used with STOP during debugging. When execution stops, you can examine and change intermediate values using Direct mode statements. Resume execution with CONT or a Direct mode GOTO, which resumes execution at a specified line number. Use CONT to continue execution after an error.

CONT is invalid if the program was edited during the break.

**EXAMPLE:**

```
10 INPUT A, B, C
20 K=A^2*5.3:L=8^21.26
30 STOP
40 M=C*K=100: PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
 30.7692
Ok
CONT
 115.9
Ok
```



## COS Function

**FORMAT:**

**COS(X)**

**PURPOSE:**

Returns the cosine of X in radians. The calculation of COS(X) is done in single-precision.

**EXAMPLE:**

```
10 X =2*COS(.4)
20 PRINT X
RUN
 1.842122
Ok
```

# CSNG Function

**FORMAT:**

**CSNG(X)**

**PURPOSE:**

Converts X to a single-precision number.

Use CINT and CDBL to convert numbers to the integer and double-precision data types.



**EXAMPLE:**

```
10 A# = 975.34217#
20 PRINT A#; CSNG(A#)
RUN
 975.34217 975.341
Ok
```

# CSRLIN Variable

**FORMAT:**

**x = CSRLIN**

**PURPOSE:**

Returns the current line or row position of the cursor in the currently active page.

3

**REMARKS:**

The value returned is from 1 to 25.

x = POS(0) returns the column location of the cursor.

Refer to the LOCATE statement to see how to set the cursor line.

**EXAMPLE:**

```
10 ROW = CSRLIN    'Record current line.
20 COL = POS(0)    'Record current column.
30 LOCATE 24,1
40 PRINT "HELLO"   'Print HELLO on last line
50 LOCATE ROW,COL 'Restore pos. to old line, column
```

# CVI, CVS, and CVD Functions

FORMAT:

**CVI( < 2-byte string > )**

**CVS( < 4-byte string > )**

**CVD( < 8-byte string > )**

PURPOSE:

Converts string values to numeric values. Converts numeric values read in from a random disk file from strings into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single-precision number. CVD converts an 8-byte string to a double-precision number.

EXAMPLE:

```
      .
      .
      .
70 FIELD #1,4 AS N$, 12 AS B$,...
80 GET #1
90 Y=CVS(N$)
      .
      .
      .
```

# DATA Statement

**FORMAT:**

**DATA** <list>

**PURPOSE:**

Stores the numeric and string constants accessed by the program's READ statement(s).

3

<list> is a list of constants containing numeric constants in any format: fixed-point, floating-point, or integer. Numeric expressions are not allowed in the list. Surround string constants with double quotation marks only if they contain commas, colons, or significant leading or trailing spaces.

**REMARKS:**

DATA statements are nonexecutable statements and can be put anywhere in a program. A DATA statement can contain as many constants, separated by commas, as fit on a line. A program can contain any number of DATA statements.

READ statements access a program's DATA statements in order by line number. Consequently, a series of DATA statements can be regarded as one continuous list of items, regardless of the location of each DATA statement.

The variable type, numeric or string, given in a READ statement must agree with the corresponding constant in the DATA statement.

DATA statements can be reread from the beginning by using the RESTORE statement.

**EXAMPLE:**

The following program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) is 3.08, and so on.

```
   .
   .
   .
80 for I=I TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
   .
   .
   .
```



This program READs string and numeric data from the DATA statement in line 30:

```
LIST
10 PRINT "CITY", "STATE", "ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", "COLORADO", 80211
40 PRINT C$,S$,Z
OK
RUN
CITY          STATE        ZIP
DENVER,       COLORADO     80211
Ok
```

# DATE$ Variable and Statement

FORMAT:

| | |
|---|---|
| **DATE$** = < **string expr** > | Sets the current date. |
| < **string expr** > = **DATE$** | Gets the current date. |

PURPOSE:

Sets or retrieves the current date.



REMARKS:

< string expr > is a valid string literal or variable.

VBASICA fetches and assigns the current date to the string variable if DATE$ is the expression in a LET or PRINT statement.

VBASICA stores the date if DATE$ is the target of a string assignment.

RULES:

1. If < string expr > is not a valid string, the "Type mismatch" error results. DATE$ retains previous values.
2. For < string var > = DATE$, DATE$ returns a 10-character string in the form "mm-dd-yyyy", where mm is the month (01 to 12), dd is the day (01 to 31), and yy is the year (1980 to 2099).
3. For DATE$ = < string expr >, < string expr > can take one of the following forms:

   "mm-dd-yy"
   "mm/dd/yy"
   "mm-dd-yyyy"
   "mm/dd/yyyy"

If any of the values are out of range or missing, VBASICA issues the "Illegal function call" error. DATE$ retains any previous date.

**EXAMPLE:**

```
DATE$ = "10-21-82"
Ok
PRINT DATE$
10-21-1982
Ok
```

## DEF FN Statement



**FORMAT:**

**DEF FN < name > [( < parlist > )] = < func def >**

**PURPOSE:**

Defines and names a user-written function.

**REMARKS:**

< name > is a legal variable name. This name, preceded by FN, becomes the name of the function.

< parlist > are the variable names in the function definition, which are replaced when VBASICA calls the function. Commas separate the items in the list.

< func def > is a one-line expression that operates the function. Variable names appearing in this expression define the function. They do not affect program variables with the same name.

A variable name in a function definition might or might not appear in the parameter list. If it does appear, VBASICA supplies the value of the parameter when the function is called. Otherwise, it uses the current value of the variable.

The variables in the parameter list represent—on a one-to-one basis—the argument variables or values given in the function call.

User-defined functions can be numeric or string. If a type is specified in the function name, VBASICA forces the value of the expression to that type before it returns it to the calling statement. A "Type mismatch" error occurs when a type specified in the function name does not match the argument type.

Execute a DEF FN statement before calling the function it defines. If you call a function before it is defined, an "Undefined user function" error occurs. DEF FN is illegal in Direct mode.



**EXAMPLE:**

In the following example, line 410 defines the function FNAB. The function is called in line 420:

```
      .
      .
410 DEF FNAB (X,Y)=X^3/Y^2
420 T=FNAB (I,J)
      .
      .
```

# DEF SEG Statement

**FORMAT:**

**DEF SEG [= <address>]**

**PURPOSE:**

Assigns the current segment address referenced by a subsequent CALL or POKE statement or by USR or PEEK functions.

**REMARKS:**

<address> is a valid numeric expression returning an unsigned integer from 0 to 65535. VBASICA saves the <address> specified for use as the segment required by the PEEK, POKE, and CALL statements.

**RULES:**

1. Any value entered outside this range results in an "Illegal function call" error. DEF SEG retains the previous value.
2. If you omit the address option, the segment used is set to VBASICA's data segment. This value is the initial default.
3. If you give the address option, it should be a value based on a 16-byte boundary. For PEEK, POKE, or CALL statements, VBASICA shifts the value left four bits to form the code segment address for the subsequent call instruction. VBASICA does not perform additional checking to ensure that the resultant segment + offset value is valid.

   Refer to the POKE statement for information on the special interpretation of segment &HFFFF.
4. **NOTE**: DEF and SEG **must** be separated by a space. Otherwise, VBASICA interprets the statement DEFSEG = 100 to mean: "assign the value 100 to the variable DEFSEG."

EXAMPLE:

```
10 DEF SEG=0  'Set segment to interrupt table
20 DEF SEG    'Restore segment to VBASICA's DS.
```

## DEFtype Statement

FORMAT:

**DEF** < **type** > < **range** >

PURPOSE:

Declares variable types as integer, single-precision, double-precision, or string.

REMARKS:

< type > is INT, SNG, DBL, or STR.

< range > is a range of letters (A–Z).

A DEF < type > statement declares that variable names beginning with the specified letter(s) are of the type specified. A type-declaration character always takes precedence over a DEF < type > statement when assigning a type to a variable.

If a program does not contain type-declaration statements, VBASICA assumes that all variables without declaration characters are single-precision variables.

**EXAMPLE:**

In this example, all variables beginning with the letters L, M, N, O, and P are double-precision variables:

```
10 DEFDBL L-P
```

This statement declares that all variables beginning with the letter A are string variables:

```
10 DEFSTR A
```

In the following statement all variables beginning with the letters I through N and W through Z are integer variables:

```
10 DEFINT I-N, W-Z
```



## DEF USR Statement

**FORMAT:**

**DEF USR[** < **digit** > **]** = < **int expr** >

**PURPOSE:**

Specifies the starting address of an assembly language subroutine.

**REMARKS:**

< digit > is any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is specified. If you omit < digit >, VBASICA assumes DEF USR.

< int expr > is the value of the starting address of the USR routine. (See Appendix E.)

Any number of DEF USR statements can appear in a program to redefine subroutine starting addresses. This allows access to as many subroutines as necessary.

**EXAMPLE:**

```
    .
    .
    .
200 DEF USR=24000
210 X=USR (Y^2/2.89)
    .
    .
    .
```

3

## DELETE Command

**FORMAT:**

**DELETE[ < line1 > ][ − < line2 > ]**

**PURPOSE:**

Deletes program lines.

**REMARKS:**

< line1 > and < line2 > are the numbers of two different program lines.

VBASICA always returns to command level after it executes a DELETE. An "Illegal function call" error occurs if < line1 > or < line2 > do not exist.

**EXAMPLE:**

This statement deletes line 40:

```
DELETE 40
```

This statement deletes lines 40 through 100, inclusive:

```
DELETE 40-100
```

This final statement deletes all lines up to and including line 40:

```
DELETE-40
```



## DIM Statement

**FORMAT:**

**DIM** < **varlist** >

**PURPOSE:**

Specifies the maximum values for array variable subscripts and allocates storage accordingly.

**REMARKS:**

< varlist > is a list of subscripted variables.

If you use an array variable name without including a DIM statement, VBASICA assumes that the maximum value of its subscript(s) is 10. If you use a subscript greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless specified otherwise with the OPTION BASE statement.

The DIM statement gives all elements of the specified arrays an initial value of zero.

**EXAMPLE:**

```
10 DIM A (20)
20 FOR I=0 TO 20
30 READ A (I)
40 NEXT I
   .
   .
   .
```

3

## DRAW Statement

**FORMAT:**

**DRAW** < **string exp** >

**PURPOSE:**

Draws a complex object as specified by < string exp > for graphics only.

**REMARKS:**

< string exp > is a string expression returning a valid formatted string, using the movement commands.

DRAW combines most of the capabilities of the other graphics statements into an easy-to-use object definition language called Graphics Macro Language (GML). A GML command is a single character within a string, optionally followed by one or more characters.

### *Movement Commands*

Each of the following movement commands begin movement from the current graphics position. This position is usually the coordinate of the last graphics point plotted with another GML command, LINE, or PSET. The current position defaults to the center of the screen when a program is run. Refer to Chapter 1.1.2 for more information on screens.

***Table 3-3: Movement Commands***

| COMMAND | ACTION |
|---|---|
| U [<n>] | Move up (scale factor * n) points |
| D [<n>] | Move down |
| L [<n>] | Move left |
| R [<n>] | Move right |
| E [<n>] | Move diagonally up and right |
| H [<n>] | Move diagonally up and left |
| G [<n>] | Move diagonally down and left |
| F [<n>] | Move diagonally down and right |
| M <x,y> | Move absolute or relative. If you precede x by + or −, VBASICA adds x and y to the current position. VBASICA connects the new point with the current position by a line. Otherwise, it draws a line from the current position to the point x,y. |

**NOTE**: The commands move one unit if you supply no argument.

The commands listed in Table 3-4 can precede any of the movement commands.

## *Table 3-4: Prefixes to Movement Commands*

| PREFIX | ACTION |
|---|---|
| B | Move but don't plot any points. |
| N | Move but return to original position when done. |
| A <n> | Set angle n. n is from 0 to 3, where 0 is zero degrees; 1 is 90; 2 is 180; and 3 is 270. VBASICA scales figures rotated 90 or 270 degrees so that they appear the same size as with 0 or 180 degrees on a screen with the standard aspect ratio of 3 to 2. |
| TA <n> | Turn angle n. The value of n is from −360 to +360. If n is positive (+), the angle turns counterclockwise. If n is negative (−), the angle turns clockwise. Values entered outside of the range −360 to +360 cause an "Illegal function call" error. This command is valid for VBASICA version 2.0 and later releases. |
| C <n> | Set attribute n. n is from 0 to 3 in medium-resolution, and 0 to 1 in high-resolution. |
| S <n> | Set scale factor. n is from 1 to 255. VBASICA multiplies the scale factor by the distances given with U, D, L, R, or relative M commands to determine the actual distance traveled. |
| X <string> | Execute substring (not supported by VBASICA compiler). You can execute a second substring from a string with this command, much like GOSUB in BASIC. You can have one string execute another, which executes a third, and so on. |
| P paint,boundary | Set figure color to paint and border color to boundary. The paint parameter is an integer expression. It chooses an attribute range for the current screen mode. In medium resolution, this color is one from the current palette (0–3), defined by the COLOR statement. In high resolution, two color attributes (0 = black and 1 = white) are available.<br><br>The boundary parameter is the border color of the figure to be filled, in the attribute range for the current screen mode. You must specify both paint and boundary, or an error results. This command does not support paint tiling, and is valid for VBASICA 2.0 and later releases.<br><br>Numeric arguments can be constants like 123 or =variable;, where variable is the name of a variable (not supported by VBASICA compiler). |

VBASICA clips points off the screen.

**EXAMPLE:**

To draw a box:

```
10 SCREEN 2   'must be in graphics mode
20 SIDE.LEN = 50   'set the length of each side
30 DRAW "U=SIDE.LEN;R=SIDE.LEN;D=SIDE.LEN;L=SIDE.LEN;"
```

To draw a triangle:

```
10 SCREEN 2   'must be in graphics mode
20 DRAW "E15;F15;L30"
```



# EDIT Command

**FORMAT:**

**EDIT** <**line number**>

**EDIT** <.>

**PURPOSE:**

With the Full Screen Editor, the EDIT statement displays the line specified and positions the cursor under the first digit of the line number. The line can then be modified using the keys described in Chapter 2.3.

**REMARKS:**

<line number> is the program line number of a line existing in the program. If no such line exists, VBASICA displays line exists, the "Undefined Line Number" error message.

<.> always gets the last line referenced by an EDIT statement, LIST command, or error message.

# END Statement

**FORMAT:**

**END**

**PURPOSE:**

Stops program execution, closes all files, and returns to command level.

3

**REMARKS:**

END statements can occur anywhere in the program. Unlike the STOP statement, a BREAK message does not appear with END. An END statement at the end of a program is optional. VBASICA always returns to the command level after END.

**EXAMPLE:**

```
520 IF K>1000 THEN END ELSE GOTO 20
```

# ENVIRON Statement

**FORMAT:**

**ENVIRON** < **string** >

**PURPOSE:**

Modifies a parameter in VBASICA's Environment String Table.

**REMARKS:**



< string > is a string expression. The value of the expression must be of the form < parameter-id > = < text >, or < parameter-id > < text >. Everything to the left of the equal sign or space is assumed to be a parameter, and everything to the right is assumed to be text.

If the parameter-id did not exist in the Environment String Table, it is appended to the end of the table. If the parameter-id exists on the table when the ENVIRON statement is executed, the existing parameter-id is deleted and the new one appended to the end of the table.

The text string is the new parameter text. If the text is a null string (""), or consists only of a semicolon (;), ENVIRON removes the existing parameter-id from the Environment String Table, and compresses the remaining body of the file.

This statement can change the PATH parameter for a child process, or pass parameters to a child by inventing a new Environment parameter. Refer to the DOS PATH command.

Errors include parameters that are not strings and an "Out of memory" when no more space can be allocated to the Environment String Table. The table usually has very little free space.

**EXAMPLE:**

The following VBASICA command creates a default PATH to the root directory on drive A:

```
PATH=A:
```

The PATH can be changed to a new value:

```
ENVIRON "PATH=A:SALES;A:ACCOUNTING"
```

A new parameter can be added to the Environment String Table:

```
ENVIRON "SESAME=PLAN"
```

The Environment String Table now contains:

```
PATH=A:SALES;A:ACCOUNTING
SESAME=PLAN
```

If you then enter:

```
ENVIRON "SESAME=;"
```

you delete SESAME, and you have a table containing:

```
PATH=A:SALES;A:ACCOUNTING
```

Refer to the ENVIRON$ function and the SHELL command.

# ENVIRON$ Function

**FORMAT:**

**ENVIRON$ ( < string parameter > )**

**ENVIRON$ ( < n > )**

**PURPOSE:**

Retrieves a parameter string from VBASICA's Environment String Table.



**REMARKS:**

< n > is an integer.

The string result returned by the ENVIRON$ function cannot exceed 255 characters. If a parameter name is specified, and it cannot be found or it has no following text, ENVIRON$ returns a null string. When the parameter name is specified, ENVIRON$ returns all the associated text that follows < parameter > = in the Environment String Table.

If the argument is numeric, ENVIRON$ returns the nth string in the Environment String Table. The string includes all the text, including the parameter name. If the nth string does not exist, ENVIRON$ returns a null string.

**EXAMPLE:**

```
100 R$=ENVIRON$("PATH")
```

Returns the current path text and stores it in string variables R$.

```
100 PRINT ENVIRON$(2)
```

Prints the second environment parameter and text on the video display.

# EOF Function

**FORMAT:**

**EOF ( < filenum > )**

**PURPOSE:**

Returns −1 (true) when the end of a sequential file is reached. Use EOF to test for end-of-file while using INPUT to avoid "Input past end" errors.



**EXAMPLE:**

```
10 OPEN "I",1,"DATA"
20 C=0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C=C+1:GOTO 30
     .
     .
     .
```

# ERASE Statement

**FORMAT:**

**ERASE** <**array list**>

**PURPOSE:**

Eliminates arrays from a program.

**REMARKS:**

Arrays can be redimensioned after they are ERASEd, or the previously allocated array space in memory can be used for other purposes. A "Duplicate Definition" error occurs if you try to redimension an array without first ERASEing it.

**EXAMPLE:**

```
    .
    .
    .
450 ERASE A, B
460 DIM B(99)
    .
    .
    .
```

# ERDEV and ERDEV$ Functions

**FORMAT:**

**ERDEV**

**ERDEV$**

**PURPOSE:**

Provides a way to obtain device-specific status information. ERDEV is an integer function that contains the error code returned by the last device to declare an error. ERDEV$ is a string function that contains the name of the device driver that generated the error.

**REMARKS:**

You cannot set these functions. ERDEV is set by the Interrupt X'24' handler when an error within DOS is detected.

ERDEV contains the INT 24 error code in the lower eight bits.

**EXAMPLE:**

If a user-installed device driver, MYLPT2, ran out of paper, and the driver's error number for that problem was 9, then

```
PRINT ERDEV, ERDEV$
```

yields

```
9  MYLPT2
```

# ERR and ERL Variables

**FORMAT:**

**ERR**

**ERL**

**REMARKS:**

The ERR and ERL variables are used in error-handling subroutines. Refer to the ON ERROR GOTO statement. ERR contains the error code for the error, and ERL contains the number of the line in which the error was detected. ERR and ERL are usually used in IF...THEN statements to direct program flow in the error-trap routine.



If the statement that caused the error was a Direct mode statement, ERL contains 65535. To test if an error occurred in a Direct statement, use the following:

```
IF 65535 = ERL THEN ...
```

Otherwise, use:

```
IF ERR = error code THEN ...
IF ERL = line number THEN ...
```

If the line number does not appear to the right of the relational operator, it cannot be renumbered by RENUM. Because ERL and ERR are reserved variables, neither can appear to the left of the equal sign in a LET (assignment) statement.

# ERROR Statement

**FORMAT:**

**ERROR** < **int expr** >

**PURPOSE:**

Simulates the occurrence of a VBASICA error or allows the user to define the error codes.

3

**REMARKS:**

< int expr > is an integer expression.

The value of < int expr > must be greater than 0 and less than 255. If the value equals that of an error message VBASICA already uses, the ERROR statement simulates the occurrence of that error, and VBASICA prints the corresponding error message.

To define your own error code, use a value greater than any used by the VBASICA error codes. (Use a very high value to prevent duplication when more error codes are added to VBASICA.) Your new error code can then be conveniently handled in an error-trap routine.

If an ERROR statement specifies a code for which no error message exists, VBASICA responds with the message "Unprintable Error." Execution of an ERROR statement for which there is no error-trap routine causes an error message to appear and execution to stop.

**EXAMPLE:**

In the following example, you type the second line; VBASICA responds with the third line:

```
LIST
10 S = 10
20 T = 5
30 ERROR S + T
40 END
Ok
RUN
String too long in 30
```



or, in Direct mode:

```
Ok
ERROR 15
String too long
Ok
```

# EXP Function

**FORMAT:**

**EXP(X)**

**PURPOSE:**

Returns e to the power of X. X must be less than or equal to 87.3365. If EXP overflows, the "Overflow" error message is displayed. Machine infinity with the appropriate sign is supplied as the result, and execution continues.

**EXAMPLE:**

```
10 X=5
20 PRINT EXP(X-1)
RUN
 54.59815
Ok
```

# FIELD Statement

**FORMAT:**

**FIELD[#]** < **filenum** > , < **width** > **AS** < **str var** > ...

**PURPOSE:**

Allocates space for variables in a random file buffer.

**REMARKS:**

< filenum > is the number under which the file was OPENed.

< width > is the number of characters to be allocated to < str var > .

< str var > is a string variable.

A FIELD statement retrieves data from a random buffer after a GET, or enters data before a PUT.

**EXAMPLE:**

```
FIELD 1, 20 AS N$, 10 AS ID$, 40 AS ADD$
```

allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does not put any data in the random file buffer.

Refer to LSET/RSET and GET.

The number of bytes allocated in a FIELD statement must not exceed the record length specified when the file was OPENed. Otherwise, a "Field overflow" error occurs. The default record length is 128.

Any number of FIELD statements can be executed for the same file. All executed statements are in effect at the same time.

Do not use a FIELDed variable name in an INPUT or LET statement if a variable name is in the random file buffer. If you execute a subsequent INPUT or LET statement that uses this variable name, the variable's pointer moves to string space.

## FILES Statement



**FORMAT:**

**FILES [ < filespec > ]**

**PURPOSE:**

Prints the names of files residing on the specified disk.

**REMARKS:**

< filespec > includes either a filename or a pathname and optional device designation.

If you omit < filespec >, VBASICA lists all the files on the currently selected drive. < filespec > is a string formula which may contain question marks (?) or asterisks (*) as wild cards. A question mark matches any single character in the filename or extension. An asterisk matches one or more characters starting at that position. The asterisk is a shorthand notation for a series of question marks. You do not need to use the asterisk to request all files on a drive. For example,

```
FILES "B:"
```

If you give a filespec with no explicit path, the current directory is the default.

**EXAMPLE:**

This statement shows all files on the current directory:

```
FILES
```

This statement shows all files with extension .BAS:

```
FILES "*.BAS"
```

This statement shows all files on drive B:

```
FILES "B:*.*"
```



The next example shows all five-letter files whose names start with "TEST" and end with the .BAS extension:

```
FILES "TEST?.BAS"
```

If SALES is a subdirectory of the current directory, this statement displays SALES < dir >. If SALES is a file in the current directory, this statement displays SALES:

```
FILES "\SALES"
```

This statement displays MARY < dir > if MARY is a subdirectory of SALES. If MARY is a file, the statement displays its name.

```
FILES "\SALES\MARY"
```

# FIX Function

**FORMAT:**

**FIX(X)**

**PURPOSE:**

Returns the truncated integer part of X.

3

**REMARKS:**

FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The major difference between FIX and INT is that FIX does not return the next lower number for negative X.

**EXAMPLES:**

```
PRINT FIX(58.75)
 58
Ok
PRINT FIX(-58.75)
 -58
Ok
```

# FOR...NEXT Statement

**FORMAT:**

```
FOR <var> =x TO y [STEP z]
    .
    .
    .
NEXT [<var>][,<var>...]
```

**PURPOSE:**



Allows a series of instructions to be performed in a loop a given number of times.

**REMARKS:**

x, y, and z are numeric expressions.

< var > is used as a counter.

The first numeric expression (x) is the initial value of the counter; the second expression (y) is the final value of the counter. VBASICA executes the program lines after the FOR statement until it encounters the NEXT statement. Then, the counter is incremented by the amount STEP specifies. VBASICA checks if the value of the counter exceeds the final value (y). If y was not exceeded, VBASICA branches back to the statement after the FOR statement and repeats the process. If y was exceeded, execution continues with the statement following the NEXT statement. The process just described is a FOR...NEXT loop.

If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is less than the initial value. VBASICA decrements the counter each time the loop executes. The loop executes until the value of the counter is less than the final value.

VBASICA skips the body of the loop if the initial value of the loop, multiplied by the sign of the step, exceeds the final value times the sign of the step.

FOR...NEXT loops can be nested; that is, one FOR...NEXT loop can be put inside another loop. Each nested loop must have a unique variable name as its counter. The NEXT statement of the inside loop must appear before that of the outside loop. If nested loops have the same end point, they can share a single NEXT statement.

If the NEXT statement references only one variable, that variable can be omitted. In this case, the NEXT statement matches the most recent FOR statement. If a VBASICA statement encounters a NEXT statement before the corresponding FOR statement, VBASICA issues a "NEXT without FOR" error message and execution stops.

**EXAMPLE:**

```
10 K=10
20 FOR I=1 TO K STEP 2
30 K=K+10
40 PRINT I;
50 PRINT K
60 NEXT I
RUN
 1  20
 3  30
 5  40
 7  50
 9  60
 Ok
```

In the following example, the initial value of the loop exceeds the final value. The loop does not execute.

```
10 I=5
20 J=0
30 FOR I=I TO J
40 PRINT I
50 NEXT I
```

In the next example, the loop executes ten times:

```
10 X=5
20 FOR X=1 TO X=5
30 PRINT X;
40 NEXT X
RUN
1  2  3  4  5  6  7  8  9  10
Ok
```

VBASICA always sets the final value of the loop variable before setting the initial value.



## FRE Function

**FORMAT:**

**FRE(0)**

**FRE(X$)**

**PURPOSE:**

Arguments to FRE are dummy arguments. FRE returns the number of memory bytes VBASICA is not using.

FRE ("") forces a garbage collection before it returns the number of unused bytes. Be patient—garbage collection can take up to 90 seconds. VBASICA does not usually collect garbage until all free memory is used. By using FRE("") periodically, you will have shorter delays for each garbage collection.

**EXAMPLE:**

```
PRINT FRE(0)
 14542
Ok
```

# GET Statement for File I/O

**FORMAT:**

**GET [#]<file number>[,<record number>]**

**PURPOSE:**

Reads a record from a random disk file into a random access buffer.



**REMARKS:**

<file number> is the number under which the file was OPENed. If <record number> is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 16,777,215.

The GET and PUT statements allow fixed-length input and output for VBASICA COM files. However, because of the low performance associated with telephone line communications, we recommend that you do not use GET and PUT for telephone communication.

**EXAMPLE:**

```
GET #1,75
```

**NOTE:** After VBASICA executes a GET statement, INPUT# and LINE INPUT# may be executed to read characters from the random file buffer. The EOF function may be used after a GET statement to see if that GET was beyond the end of file marker.

# GET and PUT Statements for COM

**FORMAT:**

**GET** < **file number** > , < **nbytes** >

**PUT** < **file number** > , < **nbytes** >

**PURPOSE:**

GET and PUT allow fixed-length I/O for COM.



**REMARKS:**

< file number > is an integer expression returning a valid file number.

< nbytes > is an integer expression returning the number of bytes to be transferred into or out of the file buffer. nbytes cannot exceed the value set by the /S: switch when VBASICA was invoked.

Because of the low performance associated with telephone line communication, you should not use GET and PUT in such applications.

**EXAMPLE:**

```
10 '***  Program to Send an ASCII file over Port A
20 INPUT "Enter ASCII file to transmit:";FILNME$
30 OPEN "COM1:9600,0,7,1" AS #1     'init PORT A
40 OPEN "Output" ,#2, FILNME$, 128 'open ASCII file
50 WHILE NOT(EOF(2))
60 GET #1      'load the file buffer with a rec
70 PUT #2,128 'send the record out port A
80 WEND
90 CLOSE #1, #2
100 END
```

# GET and PUT Statements for Graphics

FORMAT:

**GET (<x1-coord>,<y1-coord>)-(<x2-coord>,<y2-coord>),<array name>**

**PUT (<x1-coord>,<y1-coord>),<array>[,<action verb>]**

PURPOSE:

Reads (GET) or writes (PUT) pixels to or from an area of the screen.

REMARKS:

<x1-coord> and <y1-coord> are numeric expressions returning a value in the integer range that specifies one corner of the rectangular area.

<x2-coord> and <y2-coord> are numeric expressions returning a value in the integer range that specifies the opposite corner of the rectangular area.

<array name> is a previously dimensioned array to receive the graphics points.

<array> is an array containing graphics information.

<action verb> is one of the following:

PSET, PRESET, AND, OR, XOR

The PUT and GET statements transfer graphics images to and from the screen. PUT and GET make animation and high-speed object motion possible in either Graphics mode.

The GET statement transfers the screen image into the array. The rectangle described by the specified points bounds the screen image. You define the rectangle in the same way as the rectangle drawn by the LINE statement using the ,B option.

The array is a place to hold the image and can be of any type except string. It must be dimensioned large enough to hold the entire image. The contents of the array after a GET are meaningless when interpreted directly, unless the array is of type integer.

The PUT statement transfers the image stored in the array onto the screen. The specified point is the coordinate of the top left corner of the image. The "Illegal function call" error results if the image to be transferred is too large to fit on the screen.



The action verb interacts the transferred image with the image already on the screen. PSET transfers the data onto the screen verbatim.

PRESET is the same as PSET except that VBASICA produces a negative image (black on white).

Use AND to transfer the image only if an image already exists under the transferred image. Use OR to superimpose the image onto the existing image.

XOR inverts the points on the screen where a point exists in the array image. This behavior is exactly like the cursor on the screen. XOR has a unique property that makes it especially useful for animation: when an image is PUT against a complex background twice, the background is restored unchanged. Thus, you can move an object around the screen without losing the background.

**NOTE**: The default action mode is XOR.

It is possible to GET an image in one mode and PUT it in another, although the effect might be unusual because of the way points are represented in each mode.

AND, OR, and XOR have the following effects on color:

AND

| array attr | screen attrib 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 2 | 0 | 0 | 2 | 2 |
| 3 | 0 | 1 | 2 | 3 |

OR

| array attr | screen attrib 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 |
| 1 | 1 | 1 | 3 | 3 |
| 2 | 2 | 3 | 2 | 3 |
| 3 | 3 | 3 | 3 | 3 |

XOR

| array attr | screen attrib 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 |
| 1 | 1 | 0 | 3 | 2 |
| 2 | 2 | 3 | 0 | 1 |
| 3 | 3 | 2 | 1 | 0 |

Animation of an object can be performed as follows:

1. PUT the object(s) on the screen.
2. Recalculate the new position of the object(s).
3. PUT the object(s) on the screen a second time at the old location(s) to remove the old image(s).
4. Return to step 1, this time PUTting the object(s) at the new location.

This type of movement leaves the background unchanged. Flicker can be decreased by minimizing the time between steps 4 and 1, and by ensuring enough time delay between steps 1 and 3. If more than one object is being animated, process all objects at once, one step at a time.

If you don't have to preserve the background, you can perform animation by using the PSET action verb. Leave a border as large or larger than the maximum distance the object will move around the image when you first get it. Thus, when the object is moved, the border effectively erases any points. This method can be faster than the method using XOR because only one PUT is required to move an object (although you must PUT a larger image).

VBASICA stores the information in the array as follows:

2 bytes giving x dimension in bits
2 bytes giving y dimension
The array data itself



The data for each row of pixels is left-justified on a byte boundary. If there are less than a multiple of 8 bits stored, the rest of the byte is filled with zeros. The required array size in bytes is the following:

4 + INT((x* < bits/pixel > + 7)/8)*y

where < bits/pixel > is 2 in screen mode 1, and 1 in screen mode 2.

The bytes per element of an array are the following:

2 for integer
4 for single precision
8 for double precision

For example, if you want to GET a 10-by-12 image into an integer array, the number of bytes required is 4 + INT((10*2 + 7)/8)*12, or 40 bytes. Thus, you need an integer array with at least 20 elements.

You can examine the x and y dimensions and the data if you use an integer array. The dimension is in element 0 of the array, and the y dimension is in element 1. Integers are stored low byte first, then high byte. The data is transferred high byte first (leftmost), then low byte.

# GOSUB...RETURN Statements

**FORMAT:**

```
GOSUB <line>
  .
  .
  .
RETURN
```

**PURPOSE:**

Branches program execution to a user-defined subroutine beginning at <line>. Control returns to the main program when the subroutine finishes executing.

**REMARKS:**

<line> is the number of the first line of the subroutine.

In a subroutine, the RETURN statement causes VBASICA to branch back to the statement immediately after the most recent GOSUB statement. A subroutine can be called any number of times in a program. You can call a subroutine from within another subroutine. Only available memory limits the nesting of subroutines.

You can use subroutines anywhere in the program, but ensure that the subroutine is easily distinguishable from the main program. You can put a STOP, END, or GOTO statement before a subroutine to direct program control around the subroutine, and to prevent entering the subroutine by accident.

**EXAMPLE:**

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT "IN";
60 PRINT "PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```



## GOTO Statement

**FORMAT:**

**GOTO** < **line** >

**PURPOSE:**

Branches unconditionally out of the normal program sequence to a specified line number.

**REMARKS:**

< line > is the line number of the statement branched to.

If < line > is an executable statement, VBASICA executes both it and any following executable statements. If < line > is a nonexecutable statement, execution starts at the first executable statement encountered after < line > .

**EXAMPLE:**

```
LIST
10 READ R
20 PRINT "R =";R,
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
Ok
RUN
R = 5           AREA = 78.5
R = 7           AREA = 153.86
R = 12          AREA = 452.16
?Out of DATA in 10
Ok
```

# HEX$ Function

**FORMAT:**

**HEX$(X)**

**PURPOSE:**

Returns a string that represents the hexadecimal value of the decimal argument. VBASICA rounds X to an integer before evaluating HEX$(X).



Use the OCT$ function for octal conversion.

**EXAMPLE:**

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS" A$ "HEXADECIMAL"
RUN
?32
 32 DECIMAL IS 20 HEXADECIMAL
Ok
```

# IF Statement

**FORMAT:**

**IF <expr> THEN <stmnt>|<line> [ELSE <stmnt>|<line>]**

**IF <expr> GOTO <line> [ELSE <stmnt>|<line>]**

**PURPOSE:**

Makes a decision regarding program flow based on the result an expression returns.



**REMARKS:**

<stmt> is the statement or statements to be executed.

<line> is the line number branched to by the IF...THEN loop.

If the result of <expr> is not zero (true), the THEN or GOTO clause is executed. THEN is followed by a line number (for branching), or by one or more statements to be executed. GOTO is always followed by a line number. If the result of <expr> is zero (false), VBASICA ignores the THEN or GOTO clause and executes the ELSE clause (if present). Execution continues with the next executable statement. VBASICA allows a comma before THEN.

IF...THEN...ELSE statements can be nested. Nesting is limited only by the length of the line.

The following example is a legal statement:

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X
THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

If the statement does not contain equal numbers of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. The following statement does not print "A < > C" when A < > B:

```
IF A=B THEN IF B=C THEN PRINT "A=C"
     ELSE PRINT "A<>C'
```

If you are using direct mode and an IF...THEN statement is followed by a line number, an "Undefined line" error results unless you already entered a statement with the same line number while in Indirect mode.

If you use IF to test equality for a value that is the result of a floating-point computation, remember that the internal representation of the value may not be exact. Perform the test against the range over which the accuracy of the value can vary.



**EXAMPLE:**

To test a computed variable A against the value 1.0, use the following:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E − 6.

The following statement GETS the record number 1 if I is not zero:

```
200 IF I THEN GET#1, I
```

In the following example a test determines if I is greater than 10 and less than 20. If I is within this range, DB is calculated and execution resumes at line 300. If I is not within the range, execution resumes at line 110.

```
100 IF (I<20)*(I>10) THEN DB=1979-1:GOTO 300
110 PRINT "OUT OF RANGE"
      .
      .
      .
```

In the following statement output goes to the screen or the line printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer; otherwise, output goes to the screen.

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

## INKEY$ Variable



**FORMAT:**

**INKEY$**

**PURPOSE:**

Returns either a one-character string containing a character read from your computer, or a null string (if no character is pending). No characters are echoed. VBASICA passes all characters through to the program except for CTRL-C, which terminates the program. (With the VBASICA compiler, CTRL-C is also passed through to the program.)

**EXAMPLE:**

```
1000 'TIMED INPUT SUBROUTINE
1010 RESPONSE$=""
1020 FOR I%=1 TO TIMELIMIT%
1030 A$=INKEY$ : IF LEN(A$)=0 THEN 1060
1040 IF ASC(A$)=13 THEN TIMEOUT%=0 : RETURN
1050 RESPONSE$=RESPONSE$+A$
1060 NEXT I%
1070 TIMEOUT%=1 : RETURN
```

# INP Function

**FORMAT:**

**INP(I)**

**PURPOSE:**

Returns the byte read from port I. INP is the complementary function to the OUT statement.

**REMARKS:**

I is a valid machine port number from 0 to 65535.

**EXAMPLE:**

The following example:

```
100 A=INP (54321)
```

is equivalent to assembly language:

```
MOV       DX,54321
 IN       AL,DX
```

# INPUT Statement

**FORMAT:**

**INPUT[;][ < "prompt" > {;,}] < varlist >**

**PURPOSE:**

Allows keyboard input during program execution.

**REMARKS:**

< "prompt" > is the text of a screen prompt.

< varlist > is a list of variables that will receive input data.

When VBASICA reaches an INPUT statement, program execution pauses and VBASICA displays a question mark to indicate that you must enter data. If < "prompt" > is included in the INPUT statement, a prompt string appears before the question mark. Execution resumes after you type the required data.

To suppress the question mark, put a comma after the prompt string instead of a semicolon.

For example, the following statement prints "ENTER BIRTHDATE" without a question mark:

```
INPUT "ENTER BIRTHDATE:",B$
```

If INPUT is followed immediately by a semicolon, then the next INPUT or PRINT statement is on the same line. VBASICA assigns the data you enter to the variables in < varlist > . The number of data items you supply must match the number of variables in the list. Separate data items by commas.

The variable list can contain numeric or string variable names, including subscripted variables. The type of each input data item must agree with the type specified by the variable name. You need not assign strings in an INPUT statement with quotation marks.

If you respond to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, for example), the message "?Redo from start" appears. Input values are not assigned until you make an acceptable response.

**EXAMPLE:**

In these examples, you must enter data before the program finishes executing:

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
RUN
?5
 5 SQUARED IS 25
Ok
LIST
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS"
30 A=PI*R^2
40 PRINT "THE AREA OF THE C    E IS" ;A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS? 7.4
THE AREA OF THE CIRCLE IS 171.9464
```

# INPUT# Statement

**FORMAT:**

**INPUT# < filenum >, < varlist >**

**PURPOSE:**

Reads data items from a sequential disk file and assigns those items to program variables.

3

**REMARKS:**

< filenum > is the number used when the file is OPENed for input.

< varlist > contains the variable names assigned to the items in the file. The variable type must match the type the variable name specifies.

INPUT# does not prompt you with a question mark. The data items in the file should appear just as they would if you were typing data in response to an INPUT statement. VBASICA ignores leading spaces, carriage returns, and linefeeds when used as part of numeric values. VBASICA assumes the first character that is not a space, carriage return, or linefeed is the start of a number. The number must end on a space, carriage return, linefeed, or comma.

VBASICA also ignores leading spaces, carriage returns, and linefeeds when scanning the sequential data file for a string item. The first character that is not a space, carriage return, or linefeed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item consists of all characters read between the first quotation mark and the second. A quoted string cannot contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and terminates on a comma, carriage return, or linefeed (or after reading 255 characters). If end-of-file is reached while a numeric or string item is being input, the item is terminated.

# INPUT$ Function

**FORMAT:**

**INPUT$(X[,[#]Y])**

**PURPOSE:**

Returns a string of X characters, from the keyboard or from file number Y. If the keyboard is used for input, no characters are echoed. All Control characters are passed through except CTRL-C, which interrupts the execution of the INPUT$ function.



**EXAMPLE:**

```
5 'LIST THE CONTENTS OF A SEQUENTIAL FILE
  IN HEXADECIMAL
10 OPEN"I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
     .
     .
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
     .
     .
     .
```

# INSTR Function

**FORMAT:**

**INSTR([I,]X$,Y$)**

**PURPOSE:**

Searches for the first occurrence of string Y$ in X$ and returns the position at which the match is found.



**REMARKS:**

I is an optional offset in the range 1 to 255. X$ and Y$ are string variables, string expressions, or string literals.

Optional offset I sets the position where the search starts. INSTR returns 0 if I is less than LEN($), X$ is null, or if Y$ cannot be found. If Y$ is null, INSTR returns I or 1. If you set I equal to 0, VBASICA returns the "Illegal function call in < line > " error message.

**EXAMPLE:**

```
10 X$ = "ABCDEB"
20 Y$ = "B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
RUN
 2   6
Ok
```

# INT Function

**FORMAT:**

**INT(X)**

**PURPOSE:**

Returns the largest integer less than or equal to X.

See also the FIX and CINT functions. Both also return integer values.



**EXAMPLES:**

```
PRINT INT(99.89)
 99
Ok

PRINT INT(-12.11)
-13
Ok
```

# IOCTL Statement

**FORMAT:**

**IOCTL [#] < filenumber > , < string >**

**PURPOSE:**

Transmits a control character or string to a device driver.

3

**REMARKS:**

IOCTL commands are usually two to three characters followed optionally by an alphanumeric argument. An IOCTL$ command string can be up to 255 bytes long.

The IOCTL statement works only if the following occur:

1. The device driver is installed.
2. The device driver states it processes IOCTL strings.
3. VBASICA performs an OPEN on a file on that device.

Most standard DOS device drivers don't process IOCTL strings. You must determine whether the specific driver can handle the command.

**EXAMPLE:**

If you want to set the page length to 66 lines per page on LPT1, follow this procedure:

```
10 OPEN "\DEV\LPT1" FOR OUTPUT AS #1
20 IOCLT$ #1, "PL66"
```

Also see the IOCTL$ function.

# IOCTL$ Function

**FORMAT:**

**IOCTL$ ([#]<filenumber>)**

**PURPOSE:**

Receives a control data string from a device driver.

**REMARKS:**



The IOCTL$ function receives acknowledgment that an IOCTL statement succeeded or failed, or obtains current status information.

IOCTL$ can ask a communications device to return the current baud rate, information on the last error, and logical line width.

The IOCTL$ function works only if the following occur:

1. The device driver is installed.
2. The device driver states that it processes IOCTL strings.
3. VBASICA performs an OPEN on a file on that device.

**EXAMPLE:**

This example tells the device that the data is raw:

```
10 OPEN "\DEV\FOO" AS #1
20 IOCTL #1, "RAW"
```

In this continuation, if the Character Driver FOO responds "false" from the raw data mode IOCTL statement, then the file is closed:

```
30 IF IOCTL$(1) = "0" THEN CLOSE 1
```

Also see the IOCTL statement.

# KEY Statement

**FORMAT:**

**KEY <key number>,<string expression>**

**KEY LIST**

**KEY ON**

**KEY OFF**

3

**PURPOSE:**

The KEY statement allows function keys to be designated as soft keys. Any one or all of the special function keys can be assigned a 15-byte string, which is input to VBASICA when the key is pressed.

Initially, the soft keys are assigned the following values:

| | |
|---|---|
| F1 = LIST | F2 = RUN |
| F3 = LOAD" | F4 = SAVE" |
| F5 = CONT(cr) | F6 = ,"LPT1:"(cr) |
| F7 = TRON(cr) | F8 = TROFF(cr) |
| F9 = KEY | F10 = SCREEN 0,0,0(cr) |

**REMARKS:**

<key number> is the key number, an expression returning an unsigned integer in the range 1 to 10.

<string expression> is the key assignment text, any valid string expression up to 15 characters in length.

KEY ON — The initial setting displays the key values on the 25th line. Displays only the first 6 characters of each value. A carriage return in the string is indicated by <.

KEY OFF — Erases the soft key display from the 25th line.

KEY LIST — Lists all ten soft key values on the screen. Displays all 15 characters of each value.

CTRL-T — Displays next set of function keys, and toggles the display off and on.

In addition to defining soft keys (F1–F10), you can trap any shifted or unshifted key by defining it with the statement:

**KEY n, CHR$ (< shift state >) + CHR$ (< scan code >)**

where n is from 15 to 20.

< shift state > is a value that corresponds to the hex value for the current shift keys. Shift state values must be in hexadecimal.

| | |
|---|---|
| Caps Lock | &H40 |
| Num Lock | &H20 |
| ALT | &H08 |
| CTRL | &H04 |
| Shift | &H01, &H02, &H03 |

You can use combinations of shift states; for example, CHR$(&H0C) represents CTRL-ALT together.

< scan code > is a value from 1 to 83 that represents the key to be trapped. It is the number of the key on the keyboard, not the ASCII value generated by pressing the key. See Appendix E.

**RULES:**

1. If the value returned for < key number > is not from 1 to 10, VBASICA displays the "Illegal function call" error. VBASICA retains the previous key string assignment.
2. The key assignment string can be 1 to 15 characters in length. If the string is longer, VBASICA assigns the first 15 characters.
3. Assigning a null string (string of length zero) to a soft key disables the function key as a soft key.

4. When a soft key is assigned, the INKEY$ function returns one character of the soft key string per invocation. If VBASICA disables a soft key, it returns the code given for that key (see Appendix E).
5. The four cursor movement keys (up, left, down, and right) are predefined as function keys 11, 12, 13, and 14, respectively. Therefore, trapping scan codes 72, 75, 77 and 80 serve no useful purpose (they are already trappables).

**EXAMPLE:**

```
50 KEY ON
```



Displays the soft keys on the 25th line.

```
200 KEY OFF
```

Erases soft key display.

```
10 KEY 1,"MENU"+CHR$(13)
```

Assigns the string 'MENU'(cr) to soft key 1. Such assignments can be used for rapid data entry. This example might be used in a program to select a menu display.

```
20 KEY 1," "
```

Erases soft key 1.

The following routine initializes the first five soft keys:

```
10 KEY OFF     'Turn off key display during init
20 DATA KEY1, KEY2, KEY3, KEY4, KEY5
30 FOR I=1 TO 5
40 READ SOFTKEYS$(I)
50 KEY I,SOFTKEYS$(I)
60 NEXT I
70 KEY ON      'now display new softkeys.
```

# KEY(n) Statement

**FORMAT:**

**KEY( < n > ) ON**

**KEY( < n > ) OFF**

**KEY( < n > ) STOP**

**PURPOSE:**

Activates and deactivates trapping of the specified key.



**REMARKS:**

< n > is a numeric expression returning a value between 1 and 20 and indicates the key to be trapped.

| | |
|---|---|
| 1–10 | Function keys 1 to 10 |
| 11 | Up arrow |
| 12 | Left arrow |
| 13 | Right arrow |
| 14 | Down arrow |
| 15–20 | Keys defined by the form: |

KEY(n),CHR$(KBflag) + CHR(scan code)

Keys 15–20 can be trapped in VBASICA 2.0 and later releases.

VBASICA must execute a KEY( < n > ) ON statement to activate trapping of function key or cursor control key activity. After KEY( < n > ) ON, if you specify a nonzero line number in the ON KEY( < n > ) statement, every time VBASICA starts a new statement it checks if the specified key was pressed. If so, it performs a GOSUB to the line number specified in the ON KEY( < n > ) statement.

If KEY( < n > ) is OFF, no trapping takes place and the event is not remembered even if it does take place.

If a KEY( < n > ) STOP statement is executed, no trapping takes place. But if the specified key is pressed, this event is remembered, and an immediate trap takes place when KEY( < n > ) ON is executed.

KEY( < n > ) ON has no effect on the display of the soft key values on the 25th line.

< n > cannot be an expression.



## KILL Command

**FORMAT:**

**KILL** **< filespec >**

**PURPOSE:**

Deletes a file from disk.

**REMARKS:**

< filespec > is a string expression for the file specification. In versions of VBASICA 2.0 and later, it can contain a path.

A "File already open" error occurs if you try to KILL a currently OPENed file.

You can use KILL with all disk file types: program files, random data files, and sequential data files.

**EXAMPLE:**

```
200 KILL "DATA 1"
```

# LEFT$ Function

**FORMAT:**

**LEFT$(X$,I)**

**PURPOSE:**

Returns a string consisting of the leftmost I characters of X$. I must be from 0 to 255. If I is greater than LEN(X$), VBASICA returns all of X$. If I is zero, VBASICA returns the null string (length zero).



**EXAMPLE:**

```
10 A$ = "BASIC PROGRAM"
20 B$ = LEFT$(A$,5)
30 PRINT B$
 BASIC
Ok
```

# LEN Function

**FORMAT:**

**LEN(X$)**

**PURPOSE:**

Returns the number of characters in X$. Counts nonprinting characters and blanks.

3

**EXAMPLE:**

```
10 X$ = "PORTLAND,OREGON"
20 PRINT LEN(X$)
 16
Ok
```

# LET Statement

**FORMAT:**

**[LET] <var> = <expr>**

**PURPOSE:**

Assigns the value of an expression to a variable.

**REMARKS:**

The word LET is optional. The equal sign suffices when assigning an expression to a variable name.

The following program fragment contains several LET statements:

```
110 LET D=12
120 LET E=12^2
130 LET F=12^4
140 LET SUM=D+E+F
      .
      .
      .
```

The same statements can also be written as the following:

```
110 D=12
120 E=12^2
130 F=12^4
140 SUM=D+E+F
      .
      .
      .
```



## LINE Statement

**FORMAT:**

**LINE [[(<x>,<y>)] -(<x1>,<y1>) [[,[<color>][,B[F][,style]]**

**PURPOSE:**

Draws or removes straight lines, rectangles, and filled rectangles for graphics only.

**REMARKS:**

<x>, <y>, <x1>, and <y1> are valid coordinates, as described in Chapter 1.1.

< color > is an expression returning a value 0 to 3, determining the color of a line. A color of 0 draws a line in the background color. See Chapter 1.1 for more information.

< style > is a 16-bit integer mask to put points on the screen. The style option is for normal lines and boxes, but cannot be used with filled boxes (BF). Using style with BF results in a syntax error. This technique is called line styling, for VBASICA 2.0 and later.

LINE (x,y)-(x1,y1) draws a line from point (x,y) to point (x1,y1).



LINE -(x1,y1) draws a line from the previous graphics cursor position to the point (x1,y1).

LINE (x1,y1)-(x2,y2),B draws a rectangle with (x1,y1) as one corner and (x2,y2) as the opposite diagonal corner.

Using the B argument replaces the following four LINE commands:

```
LINE (x1,y1)-(x2,y1)
LINE (x1,y1)-(x1,y2)
LINE (x2,y1)-(x2,y2)
LINE (x1,y2)-(x2,y2)
```

,BF draws the same rectangle as ,B but also fills in the interior points with the selected attribute.

LINE (x,y)-(x1,y1),BF draws a rectangle and fills the entire rectangle. In VBASICA 2.0 and later, out-of-range coordinates are clipped.

**EXAMPLE:**

Draw lines continuously using random attribute:

```
10 CLS
20 LINE -(RND*799,RND*399),INT(RND*4)
30 GO TO 20
```

Draw alternating pattern—line on, line off:

```
10 FOR X = 0 TO 799
20 LINE (X,0)-(X,399),X AND 1
30 NEXT
```

Draw lines continuously, using random attribute, and filling the rectangles:

```
10 CLS
20 LINE -(RND*799,RND*399),RND*2,bf
30 GO TO 20
```



## LINE INPUT Statement

**FORMAT:**

**LINE INPUT[;][" < prompt > ";] < str var >**

**PURPOSE:**

Inputs an entire line (up to 254 characters) to a string variable without using delimiters.

**REMARKS:**

The prompt is a string literal that appears on your screen before input is accepted. A question mark appears only if it is part of the prompt string. VBASICA assigns everything you type from the end of the prompt until you press the Return key to < str var > . If VBASICA encounters a linefeed/carriage return sequence (in this order only) it echoes both characters. However, VBASICA ignores the carriage return, puts the linefeed into < str var > , and data input continues.

If LINE INPUT is followed by a semicolon, then the next print or input statement is put on the same line, even if you press Return.

A LINE INPUT is bypassed if you type CTRL-C. VBASICA returns to command level and the "Ok" prompt appears. Use the CONT statement to resume execution at the LINE INPUT.

## LINE INPUT# Statement

**FORMAT:**



**LINE INPUT# < filenum >, < str var >**

**PURPOSE:**

Reads all characters in a sequential file until it reaches a carriage return. The command then skips over the carriage return/linefeed sequence and stops. The next LINE INPUT# reads all characters up to the next carriage return. Any linefeed/carriage return sequences encountered are preserved.

**REMARKS:**

< filenum > is the number under which the file is opened.

< str var > is the variable name to which the line is to be assigned.

LINE INPUT# is useful when each line of a data file is broken into fields, or if a VBASICA program saved in ASCII mode is being read as data by another program.